The Addison Wesley Signature Series

A Mike Cohn Signature Book

# Developer Testing

Building Quality into Software

Alexander Tarlinder

*Forewords by* Jeff Langr
*and* Lisa Crispin

# Developer Testing

*This page intentionally left blank*

# Developer Testing

## Building Quality into Software

Alexander Tarlinder

Addison-Wesley

Boston • Columbus • Indianapolis • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Library of Congress Control Number: 2016944434

ISBN-13: 978-0-13-429106-2
ISBN-10: 0-13-429106-9
Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
1 16

*To my grandfather Romuald, who taught me about books.*

*This page intentionally left blank*

# CONTENTS

*This page intentionally left blank*

# Foreword by Jeff Langr

Ten years ago, I became the manager and tech lead for a small development team at a local, small start-up after spending some months developing for them. The software was an almost prototypically mired mess of convoluted logic and difficult defects. On taking the leadership role, I began to promote ideas of test-driven development (TDD) in an attempt to improve the code quality. Most of the developers were at least willing to listen, and a couple eventually embraced TDD.

One developer, however, quit two days later without saying a word to me. I was told that he said something to the effect that "I'm never going to write a test, that's not my job as a programmer." I was initially concerned that I'd been too eager (though I'd never insisted on anything, just attempted to educate). I no longer felt guilty after seeing the absolute nightmare that was his code, though.

Somewhat later, one of the testers complained to me about another developer—a consultant with many years of experience—who continually submitted defect-riddled code to our QA team. "It's my job to write the code; it's their job to find the problems with it." No amount of discussion was going to convince this gentleman that he needed to make any effort to test his code.

Still later and on the same codebase, I ended up shipping an embarrassing defect that the testers failed to catch—despite my efforts to ensure that the units were well tested. A bit of change to some server code and an overlooked flipping of a boolean value meant that the client—a high-security chat application—no longer rang the bell on an incoming message. We didn't have comprehensive enough end-to-end tests needed to catch the problem.

Developer tests are tools. They're not there to make your manager happy—if that's all they were, I, too, would find a way to skip out on creating them. Tests are tools that give you the confidence to ship, whether to an end customer or to the QA team.

Thankfully, 10 years on, most developers have learned that it's indeed their job to test their own code. Few of you will embark on an interview where some form of developer testing isn't discussed. Expectations are that you're a software development professional, and part of being a professional is crafting a high-quality product. Ten years on, I'd squash any notions of hiring someone who thought they didn't have to test their own code.

Developer testing is no longer as simple as "just do TDD," or "write some integration tests," however. There are many aspects of testing that a true developer must embrace in order to deliver correct, high-quality software. And while you can find a good book on TDD or a good book on combinatorial testing, *Developer Testing:*

*Building Quality into Software* overviews the essentials in one place. Alexander surveys the world of testing to clarify the numerous kinds of developer tests, weighing in on the relative merits of each and providing you with indispensable tips for success.

In *Developer Testing*, Alexander first presents a case for the kinds of tests you need to focus on. He discusses overlooked but useful concepts such as programming by contract. He teaches what it takes to design code that can easily be tested. And he emphasizes two of my favorite goals: constructing highly readable specification-based tests that retain high documentation value, and eliminating the various flavors of duplication—one of the biggest enemies to quality systems. He wraps up the topic of unit testing with a pragmatic, balanced approach to TDD, presenting both classical and mockist TDD techniques.

But wait! There's more: In Chapter 18, "Beyond Unit Testing," Alexander provides as extensive a discussion as you could expect in one chapter on the murky world of developer tests that fall outside the range of unit tests. Designing these tests to be stable, useful, and sustainable is quite the challenge. *Developer Testing* doesn't disappoint, again supplying abundant hard-earned wisdom on how to best tackle the topic.

I enjoyed working through *Developer Testing* and found that it got even better as it went along, as Alexander worked through the meaty coding parts. It's hard to come up with good examples that keep the reader engaged and frustration free, and Alexander succeeds masterfully with his examples. I think you'll enjoy the book too, and you'll also thank yourself for getting a foundation of the testing skills that are critical to your continued career growth.

# Foreword by Lisa Crispin

The subtitle says it all—"Building Quality into Software." We've always known that we can't test quality in by testing after coding is "done." Quality has to be baked in. To do that, the entire delivery team, including developers, has to start building each feature by thinking about how to test it. In successful teams, every team member has an agile testing mind-set. They work with the delivery and customer teams to understand what the customers need to be successful. They focus on preventing, rather than finding, defects. They find the simplest solutions that provide the right value.

In my experience, even teams with experienced professional testers need developers who understand testing. They need to be able to talk with designers, product experts, testers, and other team members to learn what each feature should do. They need to design testable code. They need to know how to use tests to guide coding, from the unit level on up. They need to know how to design test code as well as—or even better than—production code, because that test code is our living documentation and our safety net. They need to know how to explore each feature they develop to learn whether it delivers the right value to customers.

I've encountered a lot of teams where developers are paid to write production code and pushed to meet deadlines. Their managers consider any time spent testing to be a waste. If these organizations have testers at all, they're considered to be less valuable contributors, and the bugs they find are logged in a defect tracking system and ignored. These teams build a mass of code that nobody understands and that is difficult to change without something breaking. Over time they generally grind to a halt under the weight of their technical debt.

I've been fortunate over the years to work with several developers who really "get" testing. They eagerly engage in conversations with business experts, designers, testers, analysts, data specialists, and others to create a shared understanding of how each feature should behave. They're comfortable pairing with testers and happily test their own work even before it's delivered to a test environment. These are happy teams that deliver solid, valuable features to their customers frequently. They can change direction quickly to accommodate new business priorities.

Testing's a vast subject, and we're all busy, so where do you start? This book delivers key testing principles and practices to help you and your team deliver the quality your customers need, in a format that lets you pick up ideas quickly. You'll learn the language of testing so you can collaborate effectively with testers, customers, and other delivery team members. Most importantly (at least to me), you'll enjoy your work a lot more and be proud of the product you help to build.

# Preface

I started writing this book four years ago with a very clear mental image of what I wanted it to be and who my readers were going to be. Four years is quite a while, and I've had to revise some of my ideas and assumptions, both in response to other work in the field and because of deepening understanding of the subject. The biggest thing that has happened during the course of those years is that the topic has become less controversial. Several recent books adopt a stance similar to this one, and there's some reassuring overlap, which I interpret as being on the right track.

## Why I Wrote This Book

I wrote this book because this was the book I should have read a decade ago! Ten years is a long time, but believe it or not, I still need this book today—although for other reasons.

Roughly 10 years ago I embarked on a journey to understand software quality. I wasn't aware of it back then; I just knew that the code that I and my colleagues wrote was full of bugs and made us sad and the customers unhappy. I was convinced that having testers execute manual routines on our software wouldn't significantly increase its quality—and time has proven me right! So I started reading everything I could find about software craftsmanship and testing, which led to two major observations.

First, to my surprise, these topics were often totally separated back then! Books about writing software seldom spoke of verifying it. Maybe they mentioned one or two testing techniques, but they tended to skip the theory part and the conceptual frameworks needed for understanding how to work systematically with testing in different contexts. That was my impression at least. On the other hand, books on testing often tended to take off in the direction of a testing process. Books on test-driven development focused on test-driven development. This applied to blogs and other online material too.

Second, writing testable code was harder than it initially appeared, not to mention turning old legacy monoliths into something that could be tested. To get a feeling for it, I had to dive deep into the areas of software craftsmanship, refactoring, legacy code, test-driven development, and unit testing. It took a lot of deliberate practice and study.

Based on these observations and my accumulated experience, I set some goals for a book project:

- Make the foundations of software testing easily accessible to developers, so that they can make informed choices about the kind and level of verification that would be the most appropriate for code they're about to ship. In my experience, many developers don't read books or blogs on testing, yet they keep asking themselves: When have I tested this enough? How many tests do I need to write? What should my test verify? I wanted these to become no-brainers.
- Demonstrate how a testing mind-set and the use of testing techniques can enrich the daily routines of software development and show how they can become a developer's second nature.
- Create a single, good enough body of knowledge on techniques for writing testable code. I realized that such a work would be far from comprehensible, especially if kept concise, but I wanted to create something that was complete enough to save the readers from plowing through thousands of pages of books and online material. I wanted to provide a "map of the territory," if you will.

This is why I should have had a book written with these goals in mind a decade ago, but why today? Hasn't the world changed? Hasn't there been any progress in the industry? And here comes the truly interesting part: this book is just as applicable today as it would have been 10 years ago. One reason is that it's relatively technology agnostic. Admittedly it is quite committed to object-oriented programming, although large parts hold true for procedural programming, and some contents apply to functional programming as well. Another reason is that progress in the field it covers hasn't been as impressive as in many others. True, today, many developers have grasped the basics of testing, and few, if any, new popular frameworks and libraries are created without testability in mind. Still, I'd argue that it's orders of magnitude easier to find a developer who's a master in writing isomorphic JavaScript applications backed by NoSQL databases running in the cloud than to find a developer who's really good at unit testing, refactoring, and, above all, who can remain calm when the going gets tough and keep applying developer testing practices in times of pressure from managers and stressed-out peers.

Being a consultant specializing in software development, training, and mentoring, I've had the privilege to work on several software development teams and to observe other teams in action. Based on these experiences, I'd say that teams and developers follow pretty much the same learning curve when it comes to quality assurance. This book is written with such a learning curve in mind, and I've done my best to help the reader overcome it and progress as fast as possible.

## Target Audience

This is a book for developers who want to write better code and who want to avoid creating bugs. It's about achieving quality in software by acknowledging testability as a primary quality attribute and adapting the development style thereafter. Readers of this book *want* to become better developers and want to understand more about software testing, but they have neither the time nor support from their peers, not to mention from their organizations.

This is not a book for beginners. It *does* explain many foundations and basic techniques, but it assumes that the reader knows how to work his development environment and build system and is no stranger to continuous integration and related tooling, like static analysis or code coverage tools. To get the most out of this book, the reader should have at least three years of experience creating software professionally. Such readers will find the book's dialogues familiar and should be able to relate to the code samples, which are all based on real code, not ideal code.

I also expect the reader to work. Even though my ambition is to make lots of information readily available, I leave the knowledge integration part to the reader. This is not a cookbook.

## About the Examples

This book contains a lot of source code. Still, my intention was never to write a programming book. I want this to be a book on principles and practices, and as such, it's natural that the code examples be written in different languages. Although I'm trying to stay true to the idioms and structure used in the various languages, I also don't want to lose the reader in fancy details specific to a single language or framework; that is, I try to keep the examples generic enough so that they can be read by anyone with a reasonable level of programming experience. At times, though, I've found this stance problematic. Some frameworks and languages are just better suited for certain constructs. At other times, I couldn't decide, and I put an alternative implementation in the appendix. The source code for the examples in the book and other related code are available on the book's companion website—http://developertesting.rocks.

## How to Read This Book

This book has been written with a very specific reader in mind: the pressed-for-time developer who needs practical information about a certain topic without having to read tons of articles, blogs, or books. Therefore, the underlying idea is that each chapter should take no more than one hour to read, preferably less. Ideally, the reader should be able to finish a chapter while commuting to work. As a consequence, the chapters are quite independent and can be read in isolation. However, starting with the first four chapters is recommended, as they lay a common ground for the rest of the material.

Here's a quick overview of the chapters:

- **Chapter 1: Developer Testing—**Explains that developers are engaged in a lot of testing activities and that they verify that their programs work, regardless of whether they call it testing or not. Developer testing is defined here.
- **Chapter 2: Testing Objectives, Styles, and Roles**—Describes different approaches to testing. The difference between testing to critique and testing to support is explained. The second half of the chapter is dedicated to describing traditional testing, agile testing, and different versions of behavior-driven development. Developer testing is placed on this map in the category of supporting testing that thrives in an agile context.
- **Chapter 3: The Testing Vocabulary**—This chapter can be seen as one big glossary. It explains the terms used in the testing community and presents some commonly used models like the matrix of test levels and test types and the agile testing quadrants. All terms are explained from a developer's point of view, and ambiguities and different interpretations of some of them are acknowledged rather than resolved.
- **Chapter 4: Testability from a Developer's Perspective**—Why should the developer care about testability? Here the case for testable software and its benefits is made. The quality attribute *testability* is broken down into observability, controllability, and smallness and explained further.
- **Chapter 5: Programming by Contract**—This chapter explains the benefits of keeping *programming by contract* in mind when developing, regardless of whether tests are being written or not. This technique formalizes responsibilities between calling code and called code, which is an important aspect of writing testable software. It also introduces the concept of assertions, which reside at the core of all testing frameworks.
- **Chapter 6: Drivers of Testability**—Some constructs in code have great impact on testability. Therefore, being able to recognize and name them is critical. This chapter explains direct and indirect input/output, state, temporal coupling, and domain-to-range ratio.
- **Chapter 7: Unit Testing**—This chapter starts by describing the fundamentals of xUnit-based testing frameworks. However, it soon moves on to more advanced topics like structuring and naming tests, proper use of assertions, constraint-based assertions, and some other technicalities of unit testing.

- **Chapter 8: Specification-based Testing Techniques**—Here the testing domain is prevalent. Fundamental testing techniques are explained from the point of view of the developer. Knowing them is essential to being able to answer the question: "How many tests do I need to write?"
- **Chapter 9: Dependencies**—Dependencies between classes, components, layers, or tiers all affect testability in different ways. This chapter is dedicated to explaining the different kinds and how to deal with them.
- **Chapter 10: Data-driven and Combinatorial Testing**—This chapter explains how to handle cases where seemingly many similar-looking tests are needed. It introduces parameterized tests and theories, which both solve this problem. It also explains generative testing, which is about taking test parameterization even further. Finally, it describes techniques used by testers to deal with combinatorial explosions of test cases.
- **Chapter 11: Almost Unit Tests**—This book relies on a definition of unit tests that disqualifies some tests that look and run almost as fast as unit tests from actually being called by that name. To emphasize the distinction, they're called "fast medium tests". They typically involve setting up a lightweight server of some kind, like a servlet container, mail server, or in-memory database. Such tests are described in this chapter.
- **Chapter 12: Test Doubles**—This chapter introduces typical test doubles like stubs, mocks, fakes, and dummies, but without using any mocking frameworks. The point is to understand test doubles without having to learn yet another framework. This chapter also describes the difference between state-based and interaction-based testing.
- **Chapter 13: Mocking Frameworks**—Here it gets very practical, as the mocking frameworks Moq, Mockito, and the test double facilities of Spock are used to create test doubles for different needs and situations—especially stubs and mocks. This chapter also includes pitfalls and antipatterns related to the use of mocking frameworks.
- **Chapter 14: Test-driven Development—Classic Style**—Here, classic test-driven development is introduced through a longer example. The example is used to illustrate the various details of the technique, such as the order in which to write tests and strategies for making them pass.
- **Chapter 15: Test-driven Development—Mockist Style**—There's more than one way to do test-driven development. In this chapter, an alternative way is described. It's applicable in cases where test driving the design of the system is more important than test driving the implementation of a single class or component.

- **Chapter 16: Duplication**—This chapter explains why code duplication is bad for testability, but sometimes a necessary evil to achieve independence and throughput. Two main categories of duplication are introduced and dissected: mechanical duplication and duplication of knowledge.
- **Chapter 17: Working with Test Code**—This chapter contains suggestions on what to do before resorting to comments in test code and when to delete tests.
- **Chapter 18: Beyond Unit Testing**—Unit testing is the foundation of developer testing, but it's just *one* piece of the puzzle. Software systems of today are often complex and require testing at various levels of abstraction and granularity. This is where integration, system, and end-to-end tests come in. This chapter introduces such tests through a series of examples and discusses their characteristics.
- **Chapter 19: Test Ideas and Heuristics**—This final chapter, on the border of being an appendix, summarizes various test heuristics and ideas from the book.

Register your copy of *Developer Testing* at informit.com for convenient access to downloads, updates, and corrections as they become available. To start the registration process, go to informit.com and log in or create an account. Enter the product ISBN (9780134291062) and click Submit. Once the process is complete, you will find any available bonus content under "Registered Products."

# Acknowledgments

Writing a book is a team effort. The author is the one who writes the text and spends the most time with it, but many people make their contributions. This book is no exception. My first thanks go to Joakim Tengstrand, an expert in software development with a unique perspective on things, but above all, my friend. He's been giving me continual and insightful feedback from very early stages of writing to the very end.

Another person who needs a special mention is Stephen Vance. He helped me by doing a very exhaustive second-pass technical review. Not only did he offer extensive and very helpful feedback, he also found many, if not all, places where I tried to make things easy for myself. In addition, he helped me broaden the book by offering alternatives and perspectives.

As a matter of fact, this entire book wouldn't exist in its present form without Lisa Crispin's help. She's helped me to get it published, and she has supported me whenever I needed it throughout the entire process. I'm honored to have her write one of the forewords. Speaking of which, Jeff Langr also deserves my deepest gratitude for writing a foreword as well and for motivating me to rewrite an important section that I had been postponing forever. Mike Cohn, whom I've never had the pleasure of meeting, has accepted this book into his series. I can't even express how grateful I am and what it means to me. Thanks!

While on the topic of publication, I really need to thank Chris Guzikowski at Addison-Wesley. He's been very professional throughout the process and, above all, supportive beyond all limits. I don't know how many e-mails I started with something akin to: "Thanks for your patience! There's this thing I need to do before handing in the manuscript . . ." During the process of finalizing the book, I've had the pleasure to work with very professional and accommodating people, who really made the end of the journey interesting, challenging, and quite fun. Many thanks to Chris Zahn, Lisa McCoy, Julie Nahil, and Rachel Paul.

My reviewers, Mikael Brodd, Max Wenzin, and Mats Henricson, have done a huge job going through the text while doing the first-pass technical review.

Carlos Blé deserves special thanks for taking me through a TDD session that ended up producing a solution quite different from the one in the chapter on TDD. It sure gave me some things to think about, and it eventually led to a rewrite of the entire chapter. Ben Kelly has helped me enormously in getting the details of the testing terminology right, and he didn't let me escape with dividing some work between developers and testers. Dan North has helped me get the details straight about BDD and ATDD. Frank Appel has helped me around the topic of unit testing and related material. His well-grounded and thorough comments really made me stop and think at times. Many thanks. Alex Moore-Niemi has widened the book's scope by providing a sidebar on types, a topic with which I'm only superficially familiar.

I'd also like to extend my thanks to Al Bagdonas, my first-pass proofreader and copy editor for his dedication to this project.

In addition, I'd like to thank other people who have helped me along the way or served as inspiration: Per Lundholm, Kristoffer Skjutare, Fredrik Lindgren, Yassal Sundman, Olle Hallin, Jörgen Damberg, Lasse Koskela, Bobby Singh Sanghera, Gojko Adzic, and Peter Franzen.

Last, but not least, I'm joining the scores of authors who thank their wives and families. Writing a book is an endeavor that requires a lot of passion, dedication, and above all, time away from the family. Teresia, thanks for your patience and support.

# About the Author

**Alexander Tarlinder** wrote his first computer program around the age of 10, sometime in the early nineties. It was a simple, text-based role-playing game for the Commodore 64. It had lots of GOTO statements and an abundance of duplicated code. Still, to him, this was the most fantastic piece of software ever conceived, and an entry point to his future career.

Twenty-five years later, Alexander still writes code and remains a developer at heart. Today, his professional career stretches over 15 years, a time during which he shouldered a variety of roles: developer, architect, project manager, ScrumMaster, tester, and agile coach. In all these roles, he has gravitated toward sustainable pace, craftsmanship, and attention to quality, and he eventually got test infected around 2005. In a way, this was inevitable, because many of his projects involved programming money somehow (in the banking and gaming industry), and he always felt that he could do more to ensure the quality of his code before handing it over to someone else.

Presently, Alexander seeks roles that allow him to influence the implementation process on a larger scale. He combines development projects with training and coaching, and he shares technical and nontechnical aspects of developer testing and quality assurance in conferences and local user groups meetings.

Chapter 4

# Testability from a Developer's Perspective

Testability means different things to different people depending on the context. From a bird's eye view, testability is linked to our prior experience of the things we want to test and our tolerance for defects: the commercial web site that we've been running for the last five years will require less testing and will be easier to test than the insulin pump that we're building for the first time. If we run a project, testability would be about obtaining the necessary information, securing resources (such as tools and environments), and having the time to perform various kinds of testing. There's also a knowledge perspective: How well do we know the product and the technology used to build it? How good are our testing skills? What's our testing strategy? Yet another take on testability would be developing an understanding of what to build by having reliable specifications and ensuring user involvement. It's hard to test anything unless we know how it's supposed to behave.[1]

Before breaking down what testability means to developers, let's look at why achieving it for software is an end in itself.

## Testable Software

Testable software encourages the existence of tests—be they manual or automatic. The more testable the software, the greater the chance that somebody will test it, that is, verify that it behaves correctly with respect to a specification or some other expectations, or explore its behavior with some specific objective in mind. Generally, people follow the path of least resistance in their work, and if testing isn't along that path, it's very likely not going to be performed (Figure 4.1).

That testable software will have a greater chance of undergoing some kind of testing may sound really obvious. Equally apparent is the fact that lack of testability, often combined with time pressure, can and does result in bug-ridden and broken software.

Whereas testable software stands on one side of the scale, The Big Ball of Mud (Foote & Yoder 1999) stands on the other. This is code that makes you suspect that

1. For an in-depth breakdown of testability, I recommend James Bach's work on the subject (2015).

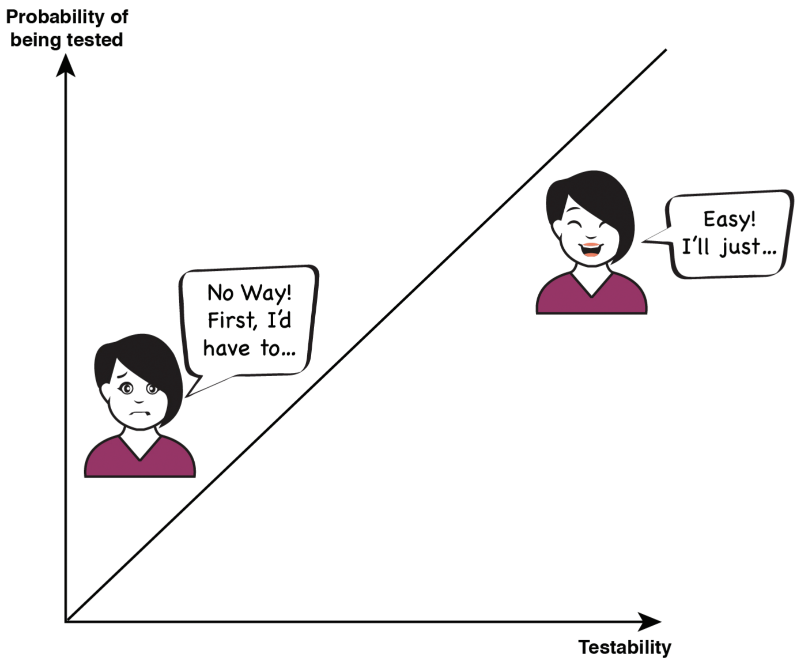

FIGURE 4.1 Is untestable software going to be tested?

the people who wrote it deliberately booby-trapped it with antitestability constructs to make your life miserable. A very real consequence of working with a system that's evolved into The Big Ball of Mud architecture is that it'll prevent you from verifying the effects of your coding efforts. For various reasons, such as convoluted configuration, unnecessary start-up time, or just the difficulty to produce a certain state or data, you may actually have a hard time executing the code you've just written, not to mention being able to write any kinds of tests for it!

For example, imagine a system that requires you to log in to a user interface (UI) and then performing a series of steps that require interacting with various graphical components and then navigating through multiple views before being able to reach the functionality you've just changed or added and want to verify. To make things more realistic (yes, this is a real-life example), further imagine that arriving at the login screen takes four minutes because of some poor design decisions that ended up having a severe impact on start-up time. As another example, imagine a batch program that has to run for 20 minutes before a certain condition is met and a specific path through the code is taken.

Honestly, how many times will you verify, or even just run, the new code if you have to enter values into a multitude of fields in a UI and click through several screens (to say nothing of waiting for the application to start up), or if you must take a coffee break every time you want to check if your batch program behaves correctly for that special almost-never-occurring edge case?

Testers approaching a system with The Big Ball of Mud architecture also face a daunting task. Their test cases will start with a long sequence of instructions about how to put the system in a state the test expects. This will be the script for how to fill in the values in the UI or how to set the system up for the 20-minute-long batch execution. Not only must the testers author that script and make it detailed enough, they must also follow it . . . many times, if they are unlucky. Brrr.

## Benefits of Testability

Apart from shielding the developers and testers from immediate misery, testable software also has some other appealing qualities.

### Its Functionality Can Be Verified

If the software is developed so that its behavior can be verified, it's easy to confirm that it supports a certain feature, behaves correctly given a certain input, adheres to a specific contract, or fulfills some nonfunctional constraint. Resolving a bug becomes a matter of locating it, changing the code, and running some tests. The opposite of this rather mechanical and predictable procedure is playing the guessing game:

Charlie: Does business rule X apply in situation Y?

Kate: Not a clue! Wasn't business rule X replaced by business rule Z in release 5.21 by the way?

Charlie: Dunno, but wasn't release 5.2 scrapped altogether? I recall that it was too slow and buggy, and that we waited for 5.4 instead.

Kate: Got me there. Not a clue.

Such discussions take place if the software's functionality isn't verifiable and is expressed as guesses instead. Lack of testability makes confirming these guesses hard and time consuming. Therefore, there's a strong probability that it won't be done.

And because it won't be done, some of the software's features will only be found in the lore and telltales of the organization. Features may "get lost" and, even worse, features may get imagined and people will start expecting them to be there, even though they never were. All this leads to "this is not a bug, it's a feature" type of arguments and blame games.

## It Comes with Fewer Surprises

Irrespective of the methodology used to run a software project, at some point somebody will want to check on its progress. How much work has been done? How much remains? Such checks needn't be very formal and don't require a written report with milestones, toll gates, or Gantt charts. In agile teams, developers will be communicating their progress at least on a daily basis in standup meetings or their equivalents.

However, estimating progress for software that has no tests (because of poor testability) ranges between best guesses and wishful thinking. A developer who believes he is "95 percent finished" with a feature has virtually no way of telling what fraction of seemingly unrelated functionality he has broken along the way and how much time it'll take to fix these regressions and the remaining "5 percent". A suite of tests makes this situation more manageable. Again, if the feature is supposedly "95 percent finished" and all tests for the new functionality pass, as well as those that exercise the rest of the system, the estimate is much more credible. Now the uncertainty is reduced to potential surprises in the remaining work, not to random regressions that may pop up anywhere in the system. Needless to say, this assumes that the codebase is indeed covered by tests that would actually break had any regression issues taken place.[2]

## It Can Be Changed

Software can always be changed. The trick is to do it safely and at a reasonable cost. Assuming that testable software implies tests, their presence allows making changes without having to worry that something—probably unrelated—will break as a side effect of that change.

Changing software that has no tests makes the average developer uncomfortable and afraid (and it should). Fear is easily observed in code. It manifests itself as duplication—the safe way to avoid breaking something that works. When doing code archaeology, we can sometimes find evidence of the following scenario:

*At some point in time, the developer needed a certain feature. Alas, there wasn't anything quite like it in the codebase. Instead of adapting an existing concept, by generalizing or parameterizing it, he took the safe route and created a parallel implementation, knowing that a bug in it would only affect the new functionality and leave the rest of the system unharmed.*

2. A slight variation of this is nicely described in the book *Pragmatic Unit Testing* by Andrew Hunt and David Thomas (2003). They plot productivity versus time for software with and without tests. The productivity is lower for software supported by tests, but it's kept constant over time. For software without tests, the initial productivity is higher, but it plummets after a while and becomes negative. Have you been there? I have.

This is but one form of duplication. In fact, the topic is intricate enough to deserve a chapter of its own.

## Why Care about Testability

Ultimately, testable software is about money and happiness. Its stakeholders can roll out new features quickly, obtain accurate estimates from the developers, and sleep well at night, because they're confident about the quality. As developers working with code every day, we, too, want to be able to feel productive, give good estimates, and be proud of the quality of our systems. We also want our job to feel fulfilling; we don't want to get stuck in eternal code-fix cycles, and, above all, we don't want our job to be repetitive and mind numbing. Unfortunately, unless our software is testable, we run that risk. Untestable software forces us to work *more* and *harder* instead of *smarter.*

### Tests Are Wasteful

by Stephen Vance

This may sound heretical in a book on developer testing and from the author of another book on code-level testing, but bear with me. Agile methods attempt to improve the software we write, or more generally, the results of our knowledge work. I'm very careful to phrase this in a way that highlights that the results are more important than the methods. If some magical Intention Machine produced the software we want without programming, this entire book would be academic. If we could achieve the results without software altogether at the same levels of speed and convenience, our entire discipline would be irrelevant. In some sense, as advanced as we are compared to the course of human history, the labor-intensive-approach trade we ply is quite primitive. Before we wither at the futility of it all, we realize we can only achieve this magical future through improvement.

Most Agile methods have some basis in the thinking that revolutionized manufacturing at the end of the twentieth century. Lean, Total Quality Management, Just-in-time, Theory of Constraints, and the Toyota Production System from the likes of Juran, Deming, Ohno, and Goldratt completely changed the state of manufacturing. Agile methods take those insights and apply them to a domain of inherent invention and variability. Although the principles must be significantly adapted, most of them still apply.

A key principle is the elimination of waste. The Toyota Production System even has three words for waste, *muda*, *mura*, and *muri*, and *mura* has at least seven subcategories captured in the acronym TIMWOOD. Much of our testing focuses on the waste of defects, but does so by incurring inventory and overprocessing.

We incur inventory waste when we invest capital (i.e., coding time) in product that has not yet derived value. Since tests are never delivered, they are eternal inventory. They are an investment with no direct return, only indirect through the reduction and possible prevention of defects.

We incur overprocessing waste by spending the extra attention required to write the tests as compared to the raw production code. The extra attention may pay off compared to the debugging time to get it right at first, the rework for the defects we don't catch, and the refamiliarization on each maintenance encounter. It is clearly additional to getting the code right naturally from the start.

The previous alternatives clearly show that tests are better than the problems they address. That just means they're the best thing we have, not the best we can do. Ultimately, we care about correctness, not tests. We need to keep looking for better ways to ensure the correctness of our software.

I haven't found the answer yet, but there are some interesting candidates.

*Domain-Specific Languages*

Domain-specific languages (DSLs) have promise. They simplify the work for their users and avoid the repetitive creation of similar code. They bring us closer to being able to say exactly what we mean in the language of the problem we are solving by encapsulating potentially complex logic in a higher-order vocabulary. If the author guarantees the correctness of the elements of the DSL, whole layers of code are correct before we try to use them.

However, good DSLs are notoriously hard to write. Arguably, almost every API we use should be a good DSL, but how many are? Creating a good DSL requires not only taking the time to understand the domain, but also playing with different models of the domain and its interactions to optimize its usability and utility. Additionally, there may be multiple characteristic usage patterns, differing levels of relevant abstractions, varying levels of user expertise, and impactful technological changes over time.

Take, for example, the Capybara acceptance test framework for Ruby, often cited as an example of a well-crafted DSL in the context of its host language. With a set of actions like `visit`, `fill_in`, `click_button` and matchers like `have_content`, it is well suited to static web pages. Under the covers, it has adapted to the rapid evolution of underlying tools like Selenium, but not without challenges at times. However, it still has difficulty dealing with the dynamic, time-dependent behaviors of single-page applications.

*Formal Methods*

Formal methods sound good. They provide formal proof of the correctness of the code. Unfortunately, we have had a hard time adapting them to larger problems, they are very labor intensive, and most programmers I've met prefer not to deal in that level of mathematical rigor. The research continues, but we're not there yet.

*Types*

Types bridge the gap between mainstream languages and formal methods in my opinion. By using a subset of formal specification, they help you ensure correctness by cleanly and compactly expressing your illegal "corner cases" in the context they can be most readily applied.

*Others*

Other approaches provide partial, complex, or laborious solutions. If you're so inclined, maybe you can find that great breakthrough. Until then, keep testing.

## Testability Defined

Testability is a quality attribute among other "ilities" like reliability, maintainability, and usability. Just like the other quality attributes, it can be broken down into more fine-grained components (Figure 4.2). Observability and controllability are the two cornerstones of testability. Without them, it's hard to say anything about correctness. The remaining components described next made it to the model based on my practical experience, although I hope that their presence isn't surprising or controversial.

When a program element (see "Program Elements") is *testable*, it means that it can be put in a known state, acted on, and then observed. Further, it means that this can be done without affecting any other program elements and without them interfering. In other words, it's about making the black box of testing somewhat transparent and adding some control levers to it.

### Program Elements

From time to time I'll be using the term *program element*. The meaning of the term depends on the context. Sometimes it's a function, sometimes a method, sometimes a class, sometimes a module, sometimes a component, or sometimes all of these things. I use the generic term to avoid clumsy sentences.

Using a catch-all term also solves the problem of emphasizing the difference between programming paradigms. Although the book favors object-oriented code, many techniques apply to procedural and functional constructs too. So instead of writing "class" and "method" everywhere, I can use "program element" and refer to "function" or "module" as well, like a C file with a bunch of related functions.

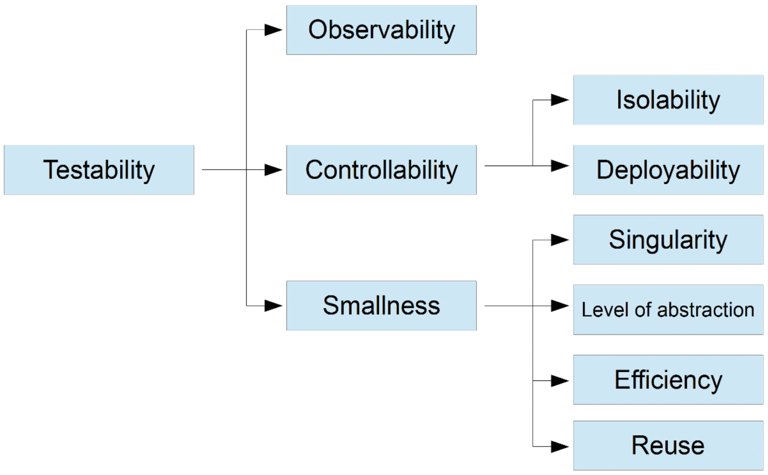

**FIGURE 4.2** The testability quality attribute decomposed.

## Observability

In order to verify that whatever action our tested program element has been subjected to has had an impact, we need to be able to observe it. The best test in the world isn't worth anything unless its effects can be seen. Software can be observed using a variety of methods. One way of classifying them is in order of increasing intrusiveness.

The obvious, but seldom sufficient, method of observation is to examine whatever output the tested program element produces. Sometimes that output is a sequence of characters, sometimes a window full of widgets, sometimes a web page, and sometimes a rising or falling signal on the pin of a chip.

Then there's output that isn't always meant for the end users. Logging statements, temporary files, lock files, and diagnostics information are all output. Such output is mostly meant for operations and other more "technical" stakeholders. Together with the user output, it provides a source of information for nonintrusive testing.

To increase observability beyond the application's obvious and less obvious output, we have to be willing to make some intrusions and modify it accordingly. Both testers and developers benefit from strategically placed observation points and various types of hooks/seams for attaching probes, changing implementations, or just peeking at the internal state of the application. Such modifications are sometimes frowned upon, as they result in injection of code with the sole purpose of increasing observability. At the last level, there's a kind of observability that's achievable only by developers. It's the ability to step through running code using a debugger. This certainly provides maximum observability at the cost of total intrusion. I don't consider this activity testing, but rather writing code. And you certainly don't want debugging to be your only means of verifying that your code works.

Too many observation points and working too far from production code may result in the appearance of *Heisenbugs*—bugs that tend to disappear when one tries to find and study them. This happens because the inspection process changes something in the program's execution. Excessive logging may, for example, hide a race condition because of the time it takes to construct and output the information to be logged.

Logging, by the way, is a double-edged sword. Although it's certainly the easiest way to increase observability, it may also destroy readability. After all, who hasn't seen methods like this:

```
void performRemoteReboot(String message) {
    if (log.isDebugEnabled()) {
        log.debug("In performRemoteReboot:" + message);
    }
    log.debug("Creating telnet client");
    TelnetClient client = new TelnetClient("192.168.1.34");
    log.debug("Logging in");
    client.login("rebooter", "secret42");
    log.debug("Rebooting");
    client.send("/sbin/shutdown -r now '" + message + "'");
    client.close();
    log.debug("done");
}
```

As developers, we need to take observability into account early. We need to think about what kind of additional output we and our testers may want and where to add more observation points.

Observability and information hiding are often at odds with each other. Many languages, most notably the object-oriented ones, have mechanisms that enable them to limit the visibility of code and data to separate the interface (function) from the implementation. In formal terms, this means that any proofs of correctness must rely solely on public properties and not on "secret" ones (Meyer 1997). On top of that, the general opinion among developers seems to be that the kind of testing that they do should be performed at the level of public interfaces. The argument is sound: if tests get coupled to internal representations and operations, they get brittle and become obsolete or won't even compile with the slightest refactoring. They no longer serve as the safety net needed to make refactoring a safe operation.

Although all of this is true, the root cause of the problem isn't really information hiding or encapsulation, but poor design and implementation, which, in turn, forces us to ask the question of the decade: *Should I test private methods?*[3]

Old systems were seldom designed with testability in mind, which means that their program elements often have multiple areas of responsibility, operate at different levels of abstraction at the same time, and exhibit high coupling and low cohesion. Because of the mess under the hood, testing specific functionality in such systems through whatever public interfaces they have (or even finding such interfaces) is a laborious and slow process. Tests, especially unit tests, become very complex because they need to set up entire "ecosystems" of seemingly unrelated dependencies to get something deep in the dragon's lair working.

In such cases we have two options. Option one is to open up the encapsulation by relaxing restrictions on accessibility to increase both observability and controllability. In Java, changing methods from private to package scoped makes them accessible to (test) code in the same package. In C++, there's the infamous `friend` keyword, which can be used to achieve roughly a similar result, and C# has its `InternalsVisibleTo` attribute.

The other option is to consider the fact that testing at a level where we need to worry about the observability of deeply buried monolithic spaghetti isn't the course of action that gives the best bang for the buck at the given moment. Higher-level tests, like system tests or integration tests, may be a better bet for old low-quality code that doesn't change that much (Vance 2013).

With well-designed *new* code, observability and information hiding shouldn't be an issue. If the code is designed with testability in mind from the start and each program element has a single area of responsibility, then it follows that all interesting abstractions and their functionality will be primary concepts in the code. In object-oriented languages this corresponds to public classes with well-defined functionality (in procedural languages, to modules or the like). Many such abstractions may be too specialized to be useful outside the system, but in context they're most meaningful and eligible for detailed developer testing. The tale in the sidebar contains some examples of this.

### Testing Encapsulated Code

Don't put yourself in the position where testing encapsulated code becomes an issue. If you're already there and can't escape in the foreseeable future, test it!

3. Or functions, or modules, or any program element, the accessibility to which is restricted by the programming language to support encapsulation.

## The Tale of the Math Package

Let’s assume that we’re setting out to build a math package with a user interface. Users will enter different expressions or equations somehow, and the software will compute the result or perform a mathematical operation like differentiation or integration.

If built iteratively in increments, possibly in a test-driven manner, the entire application may initially start in a single class or module, which will do everything: accept input, parse it, evaluate it, and eventually output the results. Such a program can easily be tested via its public interface, which would be somewhere around accepting unparsed input and returning the results of the computation. Maybe like so:

```
DisplayableResult evaluate(String userInput)
```

However, as the code grows, new program elements will be introduced behind this public interface. First a parser may appear, then something that evaluates the parsed input, then a bunch of specialized math functions, and finally a module that presents the output somehow—either graphically or using some clever notation. As all these building blocks come into existence, testing them through only the first public entry point becomes ceremonious, because they’re standalone abstractions with well-defined behavior. Consequently, all of them operate on their own data types and domains, which have their own boundary values and equivalence partitions (see Chapter 8, “Specification-based Testing Techniques”) and their own kind of error and exception handling. Ergo, they need their share of tests. Such tests will be much simpler than the ones starting at the boundary of the public interface, because they’ll hit the targeted functionality using its own domains and abstractions. Thus, a parsing module will be tested using strings as input and verified against some tree-like structure that represents the expression, whereas an evaluation module may be tested using this tree-like representation and returning something similar. If the underlying math library contains a tailor-made implementation of prime number factorization, that, too, will need specific testing.

If built with some degree of upfront design (be it detailed or rough), that design will reveal some interesting actors, like the parser or the evaluation engine, and their interfaces from the start. At this stage it will be apparent that these actors need to work together correctly, but also exhibit individual correctness. Enter tests of nonpublic behavior . . .

So what happens if, let's say, the parsing code is replaced with a third-party implementation? Numerous tests will be worthless, because the new component happens to be both well renowned for its stability and correctness and well tested. This wouldn't have happened if all tests targeted the initial public interface. Well, this is the "soft" in software—it changes. The tests that are going to get thrown away once secured the functionality of the parser, given its capabilities and implementation. The new parsing component comes with new capabilities, and certainly a new implementation, so some tests will no longer be relevant.

## Controllability

Controllability is the ability to put something in a specific state and is of paramount importance to any kind of testing because it leads to *reproducibility*. As developers, we like to deal with determinism. We like things to happen the same way every time, or at least in a way that we understand. When we get a bug report, we want to be able to reproduce the bug so that we may understand under what conditions it occurs. Given that understanding, we can fix it. The ability to reproduce a given condition in a system, component, or class depends on the ability to isolate it and manipulate its internal state.

Dealing with state is complex enough to mandate a section of its own. For now, we can safely assume that too much state turns reproducibility, and hence controllability, into a real pain. But what is *state*? In this context, state simply refers to whatever data we need to provide in order to set the system up for testing. In practice, state isn't only about data. To get a system into a certain state, we usually have to set up some data and execute some of the system's functions, which in turn will act on the data and lead to the desired state.

Different test types require different amounts of state. A unit test for a class that takes a string as a parameter in its constructor and prints it on the screen when a certain method is called has little state. On the other hand, if we need to set up thousands of fake transactions in a database to test aggregation of cumulative discounts, then that would qualify as a great deal of state.

### Deployability

Before the advent of DevOps, deployability seldom made it to the top five quality attributes to consider when implementing a system. Think about the time you were in a large corporation that deployed its huge monolith to a commercial application server. Was the process easy? Deployability is a measure of the amount of work needed to deploy the system, most notably, into production. To get a rough feeling for it, ask:

"How long does it take to get a change that affects one line of code into production?" (Poppendieck & Poppendieck 2006).

Deployability affects the developers' ability to run their code in a production-like environment. Let's say that a chunk of code passes its unit tests and all other tests on the developer's machine. Now it's time to see if the code actually works as expected in an environment that has more data, more integrations, and more complexity (like a good production-like test environment should have). This is a critical point. If deploying a new version of the system is complicated and prone to error or takes too much time, it won't be done. A typical process that illustrates this problem is manual deployment based on a list of instructions. Common traits of deployment instructions are that they're old, they contain some nonobvious steps that may not be relevant at all, and despite their apparent level of detail, they still require a large amount of tacit knowledge. Furthermore, they describe a process that's complex enough to be quite error prone.

### Manual Deployment Instructions

A list of instructions for manual deployment is a scary relic from the past, and it can break even the toughest of us. It's a sequence of steps written probably five or more years ago, detailing the procedure to manually deploy a system. It may look something like this:

1. Log in to prod.mycompany.com using ssh with user `root`, password `secret123`.
2. Navigate to the application server directory:

   ```
   cd /data/opt/extras/appserver/jboss
   ```
3. Stop the server by running the following:

   ```
   ./stop_server_v1_7.sh
   ```
4. On your local machine, run the build script:

   ```
   cd c:\projects\killerapp, ant package
   ```
5. Use WinSCP version 1.32 to copy killerapp.ear to the deployment directory.
6. Remove the temporary files in `/tmp/killerapp`.
7. Clear the application cache:

   ```
   rm -rf server/killerapp/cache*)
   ```
8. More steps . . .

Being unable to deploy painlessly often punishes the developers in the end. If deployment is too complicated and too time consuming, or perceived as such, they may stop verifying that their code runs in environments that are different from their development machines. If this starts happening, they end up in the good-old "it works on my machine" argument, and it *never* makes them look good, like in this argument between Tracy the Tester and David the Developer:

Tracy: I tried to run the routine for verifying postal codes in Norway. When I entered an invalid code, nothing happened.

David: All my unit tests are green and I even ran the integration tests!

Tracy: Great! But I expected an error message from the system, or at least some kind of reaction.

David: But really, look at my screen! I get an error message when entering an invalid postal code. I have a Norwegian postal code in my database.

Tracy: I notice that you're running build 273 while the test environment runs 269. What happened?

David: Well . . . I didn't deploy! It would take me half a day to do it! I'd have to add a column to the database and then manually dump the data for Norway. Then I'd have to copy the six artifacts that make up the system to the application server, but before doing that I'd have to rebuild three of them. . . . I forgot to run the thing because I wanted to finish it!

The bottom line is that developers are not to consider themselves finished with their code until they've executed it in an environment that resembles the actual production environment.

Poor deployability has other adverse effects as well. For example, when preparing a demo at the end of an iteration, a team can get totally stressed out if getting the last-minute fixes to the demo environment is a lengthy process because of a manual procedure.

Last, but not least, struggling with unpredictable deployment also makes critical bug fixes difficult. I don't encourage making quick changes that have to be made in a very short time frame, but sometimes you encounter critical bugs in production and they have to be fixed immediately. In such situations, you don't want to think about how hard it's going to get the fix out—you just want to squash the bug.

### What about Automated Deployment?

One way to ensure good deployability is to commit to continuous integration and then adapt the techniques described in the book *Continuous Delivery*. Its authors often repeat: "If it's painful, do it more often" (Humble & Farley 2010), and this certainly refers to the deployment process, which should be automated.

### Isolability

Isolability, modularity, low coupling—in this context, they're all different sides of the same coin. There are many names for this property, but regardless of the name, it's about being able to isolate the program element under test—be it a function, class, web service, or an entire system.

Isolability is a desirable property from both a developer's and a tester's point of view. In modular systems, related concepts are grouped together, and changes don't ripple across the entire system. On the other hand, components with lots of dependencies are not only difficult to modify, but also difficult to test. Their tests will require much setup, often of seemingly unrelated dependencies, and their interactions with the outside world will be artificial and hard to make sense of.

Isolability applies at all levels of a system. On the class level, isolability can be described in terms of *fan-out*, that is, the number of outgoing dependencies on other classes. A useful design rule of thumb is trying to achieve a low fan-out. In fact, high fan-out is often considered bad design (Borysowich 2007). Unit testing classes with high fan-out is cumbersome because of the number of test doubles needed to isolate the class from all collaborators.

Poor isolability at the component level may manifest itself as difficulty setting up its surrounding environment. The component may be coupled to other components by various communication protocols such as SOAP or connected in more indirect ways such as queues or message buses. Putting such a component under test may require that parts of it be reimplemented to make the integration points interchangeable for stubs. In some unfortunate cases, this cannot be done, and testing such a component may require that an entire middleware package be set up just to make it testable.

Systems with poor isolability suffer from the sum of poorness of their individual components. So if a system is composed of one component that makes use of an enterprise-wide message bus, another component that requires a very specific directory layout on the production server (because it won't even run anywhere else), and a third that requires some web services at specific locations, you're in for a treat.

## Smallness

The smaller the software, the better the testability, because there's less to test. Simply put, there are fewer moving parts that need to be controlled and observed, to stay consistent with this chapter's terminology. Smallness primarily translates into the quantity of tests needed to cover the software to achieve a sufficient degree of confidence. But what exactly about the software should be "small"? From a testability perspective, two properties matter the most: the number of features and the size of the codebase. They both drive different aspects of testing.

Feature-richness drives testing from both a black box and a white box perspective. Each feature somehow needs to be tested and verified from the perspective of the user. This typically requires a mix of manual testing and automated high-level tests like end-to-end tests or system tests. In addition, low-level tests are required to secure the building blocks that comprise all the features. Each new feature brings additional complexity to the table and increases the potential for unfortunate and unforeseen interactions with existing features. This implies that there are clear incentives to keep down the number of features in software, which includes removing unused ones.

A codebase's smallness is a bit trickier, because it depends on a number of factors. These factors aren't related to the number of features, which means that they're seldom observable from a black box perspective, but they may place a lot of burden on the shoulders of the developer. In short, white box testing is driven by the size of the codebase. The following sections describe properties that can make developer testing cumbersome without rewarding the effort from the feature point of view.

## Singularity

If something is singular, there's only one instance of it. In systems with high singularity, every behavior and piece of data have a single source of truth. Whenever we want to make a change, we make it in one place. In the book *The Pragmatic Programmer*, this has been formulated as the DRY principle: Don't Repeat Yourself (Hunt & Thomas 1999).

Testing a system where singularity has been neglected is quite hard, especially from a black box perspective. Suppose, for example, that you were to test the copy/paste functionality of an editor. Such functionality is normally accessible in three ways: from a menu, by right-clicking, and by using a keyboard shortcut. If you approached this as a black box test while having a limited time constraint, you might have been satisfied with testing only one of these three ways. You'd assume that the others would work by analogy. Unfortunately, if this particular functionality had been implemented by two different developers on two different occasions, then you wouldn't be able to assume that both are working properly.

| The tester sees . . . | The developer implemented . . . |
|---|---|
| Copy | `EditorUtil.copy` |
| | `currentEditorPanel.performCopy` |
| | A third version? |

This example is a bit simplistic, but this scenario is very common in systems that have been developed by different generations of developers (which is true of pretty much every system that's been in use for a while). Systems with poor singularity appear confusing and frustrating to their users, who report a bug and expect it to be fixed. However, when they perform an action similar to the one that triggered the bug by using a different command or accessing it from another part of the system, the problem is back! From their perspective, the system should behave consistently, and explaining why the bug has been fixed in two out of three places inspires confidence in neither the system nor the developers' ability.

To a developer, nonsingularity—duplication—presents itself as the activity of implementing or changing the same data or behavior multiple times to achieve a single result. With that comes maintaining multiple instances of test code and making sure that all contracts and behavior are consistent.

## Level of Abstraction

The level of abstraction is determined by the choice of programming language and frameworks. If they do the majority of the heavy lifting, the code can get both smaller and simpler. At the extremes lie the alternatives of implementing a modern application in assembly language or a high-level language, possibly backed by a few frameworks. But there's no need to go to the extremes to find examples. Replacing thread primitives with thread libraries, making use of proper abstractions in object-oriented languages (rather than strings, integers, or lists), and working with web frameworks instead of implementing Front Controllers[4] and parsing URLs by hand are all examples of raising the level of abstraction. For certain types of problems and constructs, employing functional or logic programming greatly raises the level of abstraction, while reducing the size of the codebase.

The choice of the programming language has a huge impact on the level of abstraction and plays a crucial role already at the level of toy programs (and scales accordingly as the complexity of the program increases). Here's a trivial program that adds its two command-line arguments together. Whereas the C version needs to worry about string-to-integer conversion and integer overflow . . .

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  int augend = atoi(argv[1]);
  int addend = atoi(argv[2]);

  // Let's hope that we don't overflow...
  printf("*drum roll* ... %d", augend + addend);
}
```

4. https://en.wikipedia.org/wiki/Front_Controller_pattern

. . . its Ruby counterpart will work just fine for large numbers while being a little more tolerant with the input as well.

```
puts "*drum roll* ... #{ARGV[0].to_i + ARGV[1].to_i}"
```

From a developer testing point of view, the former program would most likely give rise to more tests, because they'd need to take overflow into account. Generally, as the level of abstraction is raised, fewer tests that cover fundamental building blocks, or the "plumbing," are needed, because such things are handled by the language or framework. The user won't see the difference, but the developer who writes the tests will.

## Efficiency

In this context, efficiency equals the ability to express intent in the programming language in an idiomatic way and making use of that language's functionality to keep the code expressive and concise. It's also about applying design patterns and best practices. Sometimes we see signs of struggle in codebases being left by developers who have fought valorously reinventing functionality already provided by the language or its libraries. You know inefficient code when you see it, right after which you delete 20 lines of it and replace them with a one-liner, which turns out idiomatic and simple.

Inefficient implementations increase the size of the codebase without providing any value. They require their tests, especially unit tests, because such tests need to cover many fundamental cases. Such cases wouldn't need testing if they were handled by functionality in the programming language or its core libraries.

## Reuse

Reuse is a close cousin of efficiency. Here, it refers to making use of third-party components to avoid reinventing the wheel. A codebase that contains in-house implementations of a distributed cache or a framework for managing configuration data in text files with periodic reloading[5] will obviously be larger than one that uses tested and working third-party implementations.

This kind of reuse reduces the need for developer tests, because the functionality isn't owned by them and doesn't need to be tested. Their job is to make sure that it's plugged in correctly, and although this, too, requires tests, they will be fewer in number.

5. Now this is a highly personal experience, but pretty much all legacy systems that I've seen have contained home-grown caches and configuration frameworks.

#### Mind Maintainability!

All of the aforementioned properties may be abused in a way that mostly hurts maintainability. Singularity may be taken to the extreme and create too tightly coupled systems. Too high a level of abstraction may turn into some kind of "meta programming." Efficiency may turn into unmotivated compactness, which hurts readability. Finally, reuse may result in pet languages and frameworks being brought in, only to lead to fragmentation.

### A Reminder about Testability

Have you ever worked on a project where you didn't know what to implement until the very last moment? Where there were no requirements or where iteration planning meetings failed to result in a shared understanding about what to implement in the upcoming two or three weeks? Where the end users weren't available?

Or maybe you weren't able to use the development environment you needed and had to make do with inferior options. Alternatively, there was this licensed tool that would have saved the day had but somebody paid for it.

Or try this: the requirements and end users were there and so was the tooling, but nobody on the team knew how to do cross-device mobile testing.

After having dissected the kind of testability the developer is exposed to the most, I'm just reminding that there are other facets of testability that we mustn't lose sight of.

## Summary

If the software is designed with testability in mind, it will more than likely be tested. When software is testable, we can verify its functionality, measure progress while developing it, and change it safely. In the end, the result is fast and reliable delivery.

Testability can be broken down into the following components:

- *Observability*—Observe the tested program element in order to verify that it actually passes the test.
- *Controllability*—Set the tested program element in a state expected by the test.
- *Smallness*—The smaller the system or program element—with respect to the number of features and the size of the codebase—the less to test.

*This page intentionally left blank*

# Index